

Article

SSCFM: Separate Signature-Based Control Flow Error Monitoring for Multi-Threaded and Multi-Core Environments

Kiho Choi , Daejin Park *  and Jeonghun Cho * 

School of Electronics Engineering, Kyungpook National University, Daegu 41566, Korea; posjkh22@gmail.com

* Correspondence: boltanut@knu.ac.kr (D.P.); jcho@ee.knu.ac.kr (J.C.); Tel.: +82-53-950-5548 (D.P.)

Received: 15 January 2019; Accepted: 29 January 2019; Published: 1 February 2019



Abstract: Soft error is a key challenge in computer systems. Without soft error mitigation, control flow error (CFE) can lead to system crash. Signature-based CFE monitoring scheme is a representative technique for detecting CFEs during runtime. However, most of the signature-based CFE monitoring schemes proposed thus far are based on a single thread. Currently, the widely used multi-threaded and multi-core environments have greatly improved the performance of the computing system, but, if these schemes are applied in these environments, performance improvement is difficult to achieve, or rather performance degradation may occur. In this paper, we propose a separate signature-based CFE monitoring (SSCFM) scheme that separates the signature update and the signature verification on the thread level. The signature update is combined with application thread and signature verification and executed on separate monitor threads, so that we can expect performance improvements in multi-threaded or multi-core environments. Furthermore, the SSCFM scheme can fully cover inter-procedural CFE not covered by many signature-based CFE monitoring schemes by using inter-procedural control flow analysis. With the proposed SSCFM scheme, the execution time overhead is reduced by approximately 26.67% on average from the SEDSR scheme, and the average CFE detection rate with SSCFM is approximately 93.69%. In addition, this paper also introduces the LLVM compiler-based SSCFM generator that makes it easy to apply the SSCFM scheme to software applications.

Keywords: software signature-based control flow error monitoring; multi-threaded and multi-core system; automatic code-generation

1. Introduction

Transient fault, or soft error, is a key challenge in computer system. Transient fault is caused by electromagnetic interferences, power glitches, or highly energized particles passing through a semiconductor device [1,2]. Since a transient fault occurs intermittently, it is different from a permanent fault, and it may cause erroneous bit-flips. The bit-flips can corrupt memory such as registers, main memory and may affect control flow of the software in execution. In particular, control flow errors (CFE) due to erroneous bit-flips can lead to system crash [3].

To mitigate soft errors, many hardware-based CFE monitoring schemes and software-based CFE monitoring schemes have been proposed. Hardware-based CFE monitoring schemes [4–7] incur low run-time overhead but require additional hardware modules or modification to commodity hardware. Software-based CFE monitoring schemes do not require additional hardware module or hardware modification. Software-based CFE monitoring schemes can be divided into CFE monitoring schemes using redundancy of instructions such as jump instructions [8–11] and signature-based CFE monitoring schemes that represent control flows as signature variables and detect CFEs [12–18].

However, the software-based CFE monitoring scheme has a problem that execution time overhead is very large because additional instructions or CFE detection code are required. In particular, CFE can occur in a wide variety of forms in inter-block CFEs and inter-procedural CFEs, and there is a trade-off between CFE detection rate and execution time overhead because more detection code is needed to handle more types of CFEs.

In this paper, we propose a signature-based CFE monitoring scheme that fully covers the inter-procedural CFE that is not covered or partially covered in previous works and that is modified from the typical signature-based CFE monitoring scheme in order to take advantage of performance improvement of execution time reduction in multi-threaded or multi-core systems. The main contribution of this paper are as follows: (1) we propose a new model of signature-based CFE monitoring scheme that take advantage of performance improvements of execution overhead reduction in a multi-threaded or multi-core environment; (2) we introduce a signature-based CFE monitoring scheme that can fully cover inter-procedural CFE; and (3) we propose and implement a code-generation framework for the proposed signature-based CFE monitoring scheme that makes it easier to apply to software application programs.

The remainder of this paper consists is organized as follows. In Section 2, related works for existing signature-based CFE monitoring schemes are reviewed. Section 3 describes the proposed separate signature-based CFE monitoring scheme, while Section 4 explores the feasibility of the proposed scheme through several virtual monitoring scenarios. Section 5 introduces a code generation framework for the proposed scheme. In Section 6, we prove the validity of the proposed scheme through experimentation. Finally, we conclude in Section 7.

2. Related Works

2.1. Signature-Based CFE Monitoring Schemes

Control flow is the determined operation sequence of software, and control flow error (CFE) is the operation state from the control flow. The control flow can be expressed as a directed graph [19], as shown in Figure 1. Each node (v_n) in the graph represents a basic block of the control flow. Each edge set (e_n) corresponding to a node (v_n) refers to a basic block-to-basic block connection set starting from the node (v_n). If node v_p is connected to node v_q , it can be expressed as $v_q \in e_p$ in relation to e_p and v_q ; it can also be expressed as $v_p = \text{prior}(v_q)$. In this case, the CFE is expressed as the state that the current node v_x is in relation of $v_x \notin \text{prior}(e_x)$. The graph is called a “control flow graph” (CFG).

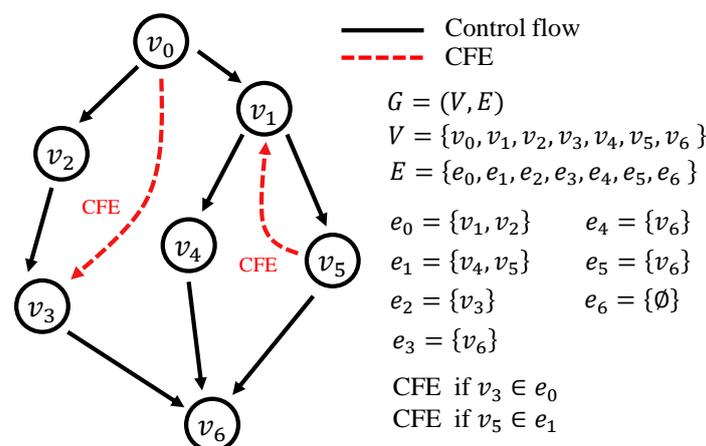


Figure 1. Control flow and CFE.

The signature-based CFE monitoring scheme is a technique for describing the control flow of software and for detecting the CFEs using signature variables, called “software signature”. In the

monitoring scheme, two routines called “signature update” and “signature verification” are alternated in basic block units. The signature update updates signature variables according to the current basic block in the CFG, and the signature verification checks that the updated signature variables do not deviate from the CFG. In other words, the signature-based CFE monitoring scheme is based on a model in which signature updates and signature verification are continuously alternated.

In the literature, many CFE monitoring schemes have been introduced in order to mitigate system failures due to CFE. Signature-based CFE monitoring is a typical software-based CFE monitoring scheme. Figure 2a shows the basic model of the existing signature-based CFE monitoring scheme, which consists of two routines: signature update $U(b_n, s)$ and signature verification $V(s, d_n)$. B_n represents the n_{th} basic block; b_n represents the signature variable in B_n and is determined at compile time; s is a signature variable and is updated during runtime; and d_n is a signature determined with $d_n = U(b_n, s)$ at compile time and it is used for comparison with the run-time signature s . The run-time signature s in B_n is updated with signature update and it is verified with signature verification in the next basic block B_{n+1} . If $V(s, d_n)$ does not meet zero, the CFE monitor judges that CFE has occurred. As the simplest case of a signature-based CFE monitoring scheme, $U(b_n, s) = b_n$ and $V(s, d_n) = s - d_n = s - b_n$ can be selected, as shown in Figure 2b.

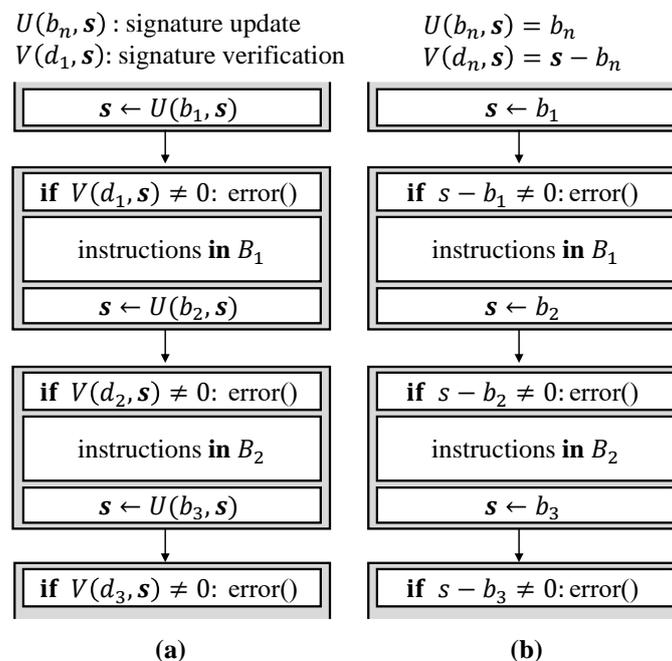


Figure 2. (a) Typical model of signature-based CFE monitoring scheme; and (b) the simplest case of signature-based CFE monitoring scheme

Most of the existing signature-based CFE monitoring schemes, such as CFCSS [16], YACCA [17], ECCA [18], RSCFC [15], SEDSR [14], SCFC [13], and RASM [12] have mainly focused on how to update and verify software signatures. They have tried to more appropriately select the number of run-time signatures and compile-time signatures, where to insert the signature update and signature verification in each basic block, and the algorithms of signature update and signature verification to improve performance through low execution overhead time and high CFE detection rate. For example, in CFCSS scheme, signature variable G is used for run-time signature, and signature variable d, D , and s are used for compile-time signature. In signature update routine, G is updated as $G = G \oplus d \oplus D$. In signature verification routine, compile-time signature s is compared with the updated G and if G is not equal with s , CFCSS judges that CFE occurs. The signature update routines and the signature verification routines are inserted sequentially at the beginning of the each basic block. In SEDSR scheme, signature variable S

is used for run-time signature, and signature variable s is used for compile-time signature. In signature update routine, S is updated as $S = s$. In signature verification routine, compile-time signature s is compared with the updated S and, if s is not equal with S , SEDSR judges that CFE occurs. In SEDSR scheme, unlike CFCSS scheme, the signature update routines are inserted in the middle of each basic block, and the signature verification routines are inserted sequentially at the beginning of the each basic block. Asghari et al. [14] claimed that simple operation without XOR or AND operation is used in signature update to lower the execution time overhead and that locating the signature update routine in the middle of the basic block will improve the CFE detection rate. In RASM scheme, signature variable $signature$ and $adjustValue$ are used for run-time signature, and signature variable $randomNumberBB$ and $subRanPreVal$ are used for compile-time signature. RASM is a two-update scheme, which is called gradual update. In gradual update, the first update is inserted at the end of the previous basic block and the second update is inserted at the beginning of current basic block. In the first signature update routine, $signature$ is updated as $signature = signature - subRanPreVal$. In the second signature update routine, $signature$ is updated as $signature = signature - adjustValue$ if $adjustValue > 0$ or $signature = signature + adjustValue$ if $adjustValue \leq 0$. In signature verification routine, compile-time signature $randomNumberBB$ is compared with the updated $signature$ and if $signature$ is not equal with $randomNumberBB$, RASM judges that CFE occurs. Vankeirsbilck et al. [12] claimed that operations such as addition and subtraction are used in the signature update and the gradual update method reduces the execution overhead and increases the CFE detection rate respectively. However, as can be seen in the examples, the previous works on signature-based CFE monitoring schemes have a very formal form and have different performance depending on the number of signatures used, the signature update/verification routine type, and the location of the routine in basic block.

2.2. Coverage of CFE Detection

CFE is caused by jump, call, and return instructions that can change the value of the program counter. Note that if a bit-flip occurs in the operand of each instruction, the program counter can be changed to any value. CFE is divided into intra-block CFE, inter-block CFE, and inter-procedural CFE. In this terminology, block refers to a basic block and procedure refers to a function. Intra-block CFE monitoring schemes detect CFE using instruction redundancy at the instruction level. Intra-block CFE monitoring schemes detect CFE by inserting redundant instructions and comparing the two instructions. However, even if the instruction is executed repeatedly, it is executed continuously, so that both the original instruction and the duplicated instruction may have errors, and the redundancy of instructions causes a lot of execution time overhead. This paper is based on the signature-based CFE monitoring scheme (included in the domain of the inter-block CFE detection or inter-procedural CFE detection). Hence, the intra-block CFE monitoring scheme through redundancy of instructions is beyond the scope of our research.

Inter-block CFE detection does not detect CFE at all instruction levels, but is a scheme for detecting control flow as a connection of basic blocks and detecting whether the CFE is out of this. Figure 3a shows the inter-block CFE generated between basic blocks. The inter-block CFE can be divided into two types: a jump to an illegal basic block and a jump to the current basic block. CFE-a illustrates a CFE caused by a jump to an illegal basic block and CFE-b illustrates a jump to the current basic block. Many previous signature-based CFE monitoring schemes [12–18] detects CFE for jump to an illegal basic block, but some schemes have not detected CFE for jump to a current basic block [13,14]. Figure 3b shows the inter-procedural CFE where the CFE occurs beyond the function boundary. The inter-procedural CFE can be divided into three types: a jump to incorrect address in a call instruction, a jump to incorrect address in a return instruction, and a jump from a basic block that has call instruction or return instruction in a function to any basic block in another function. CFE-c illustrates a jump to incorrect address in a call instruction, CFE-d illustrates a jump to incorrect address in a return instruction and CFE-e illustrates a jump from a basic block that has call instruction or return instruction in a function to any basic block in another function.

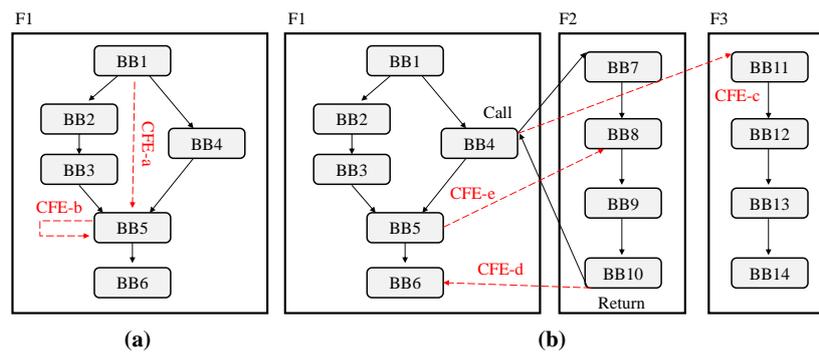


Figure 3. (a) Cases of inter-block CFE; and (b) cases of inter-procedural CFE.

Many of the previous works [12–18] do not mention inter-procedural CFE and do not address it. To deal with the inter-procedural CFE caused by a jump to incorrect address in a call instruction (CFE-c) and a jump to incorrect address in a return instruction (CFE-d), some previous works give a signature representing the function such as function identifier (FID) [20] and global signature register (GSR) [21], update it before the call instruction, and verify after function return. However, it seems that there is no proposed signature-based CFE monitoring scheme to fully detect inter-procedural CFEs. This is because a function can be called often in a functional language such as C/C++, Java, and Python. In the case of inter-block CFE detection, the connectivity between basic blocks can be represented by a maximum of two at the conditional branch. However, since a function can be called often depending on the software application, it is difficult to express the connectivities of function call/return with signature update routines and signature verification routines of existing signature-based CFE monitoring schemes.

In this paper, we propose a signature-based CFE monitoring scheme that can fully detect inter-procedural CFE as well as inter-block CFE. Through a novel concept of function stack routine that handles function calls and returns, inter-procedural CFE caused by a jump to an incorrect address in call/return instruction is detected. In addition, through the characteristics of the signature queue, inter-procedural CFE caused a jump from a basic block that does not have a call instruction or return instruction in a function to any basic block in another function is detected.

2.3. Granularity of CFE Detection

Depending on the granularity of the CFE detection, the signature-based CFE monitoring scheme is divided into two: fine-grained CFE monitoring and coarse-grained CFE monitoring schemes. The initially proposed signature-based CFE monitoring schemes were mostly fine-grained CFE monitoring schemes. A signature is assigned to each basic block and CFE monitoring is performed on connectivity of all basic blocks. The advantage of fine-grained CFE monitoring is that it is highly accurate. This is because the CFE detection code is inserted for all basic blocks. However, this causes a problem of high execution time overhead.

In recent works, coarse-grained CFE schemes using dominator [22] or super nodes [23] have been proposed to reduce execution time overhead by detecting only CFEs between dominator or super node. After considering the connectivity of the basic blocks, the dominator is selected among the basic blocks and the super node is created by combining the basic blocks. After considering the connectivity of the basic blocks, the dominator is selected among the basic blocks and the super node is created by combining the basic blocks. However, we think it has the same semantic as increasing the size of the basic block in terms of inter-block CFE detection. Therefore, this can lead to an increase in CFE within the block instead of obtaining a reduction in execution time overhead. In this paper, we propose a scheme that can reduce execution time overhead in multi-threaded or multi-core environments while following a fine-grained CFE monitoring scheme that monitors connectivity of all basic blocks.

2.4. Multi-Threading Technique and Multi-Core Environments

The multi-threading technique [24] and parallelism technique in a multi-core environment [25] lead to overall improvement in system performance and are widely used in computer systems. This technique enables maximum resource utilization for a given processor, and the parallelism technique in a multi-core environment increases the number of processors in operation at once, enabling parallel processing. However, if the previous signature-based CFE monitoring schemes are applied to a system using this multi-threading technique or a multi-core system, execution time overhead will be further increased due to context switching with other threads or other scheduling routines. This is because the previous signature-based CFE monitoring schemes are based on a single thread. In this way, previously proposed schemes that is based on a single thread can degrade performance in multi-threaded or multi-core environments. The CRDC/CRMC [26] scheme proposed in the CFE recovery scheme also mentions this.

In software-based CFE monitoring schemes, there has been an attempt to reduce the execution time overhead by utilizing a multi-threaded or multi-core system. COMET [9] is a scheme to detect CFE through redundancy of instructions. It makes redundant instructions operate in monitor threads to reduce overhead. However, the previous signature-based CFE monitoring schemes tightly combine a target software application with the CFE monitoring code on a single thread. This makes it difficult to take advantage of multi-threading technique or multi-core environments. To do so, it is necessary to separate the tightly combined target software application and CFE monitor code into more than two independent routines. In our work, we selected the signature update and the signature verification as separate routines. In our proposed signature-based CFE monitoring scheme, we modify the typical signature-based CFE monitoring scheme, leading to a performance improvement in the reduction of execution time overhead in multi-threaded or multi-core environments. The typical signature-based CFE monitoring scheme, consisting of signature update and signature verification, is separated into thread levels, and each signature update and signature verification is performed in separate threads, enabling to be applied in multi-threaded and multi-core environments.

3. Separate Signature-Based CFE Monitoring Scheme

3.1. Separation of Signature Update and Signature Verification

In this paper, based on the fact that the multi-threading technique and multi-core parallelization can improve performance, we propose a separate signature-based CFE monitoring (SSCFM) scheme that can be applied to multi-threaded or multi-core environments. For the purpose of distinguishing the existing signature-based CFE monitoring schemes from the proposed SSCFM scheme, the existing signature-based CFE monitoring schemes are hereafter referred to as “non-SSCFM” schemes.

The previously proposed non-SSCFM schemes tightly combine a target application with the CFE monitoring code on a single thread. This makes it difficult to take advantage of multi-core environments and environments. Hence, it is necessary to separate the tightly combined CFE-detectable application into more than two independent routines. The proposed SSCFM scheme treats the signature update and the signature verification as separate routines. Figure 4 describes the typical model of the non-SSCFM scheme and the proposed SSCFM scheme. In the non-SSCFM scheme, the signature update and signature verification are alternately executed in each basic block. However, in the proposed SSCFM scheme, the two routines perform separately on the thread level. The signature update, which is called “signature enqueue” in the SSCFM scheme, is executed in the “application thread” on which the monitored application runs. The signature verification is executed in a separately generated thread for verification. Hereafter, we will call the separate thread for signature verification the “monitor thread”.

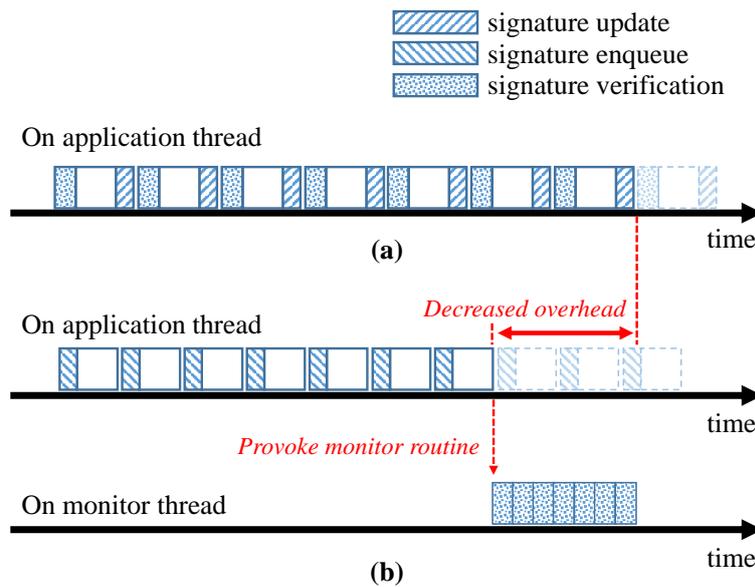


Figure 4. (a) Typical non-SSCFM scheme; and (b) the proposed SSCFM scheme.

3.2. Signature Queue

The run-time signatures in the non-SSCFM schemes have shared characteristics based on the fact that signature update is performed by overwriting in each basic block. In the proposed SSCFM scheme, the concept of a “signature queue” is introduced so that signature verification is performed collectively in monitor threads. The signature queue can be thought of as a run-time signature in the format of a queue with no shared characteristics. The signature queue is filled with the updated run-time signature in each basic block. In other words, the signature queue is updated in each basic block with a signature enqueue described below. The signature queue’s non-shared characteristics may lead to additional effects in solving the problem of CFE detection rate degradation due to run-time signature sharing in existing non-SSCFM schemes.

Figure 5 shows two cases of CFEs that may not be found due to the shared characteristics of the run-time signature in the typical non-SSCFM schemes: update after update and update after verification. This is because the run-time signature is overwritten in each basic block in the signature update. In the prior signature-based monitoring schemes, two-staged signature updates in ECCA [17] and RASM [12] or two-staged signature verification in YACCA [16] are proposed in order to solve the problem. However, these multi-staged update/verification approaches cause additional calculations and hence increase the execution time overhead.

3.3. Signature Enqueue

In our work, we introduce a new concept of “signature queue” to collect information of connectivity between basic blocks and collectively verify in separate monitor routines. In the existing non-SSCFM schemes, XOR operation [16,20,27] and AND operation [17] are generally used for signature update to indicate connectivity information between basic blocks during runtime [15,18]. However, in our proposed SSCFM scheme, the signature enqueue performs the same semantic role of updating the run-time signature as the signature update. The signature queue is responsible for putting the compile-time signatures of each basic block into the signature queue. SSCFM scheme does not use XOR operation or AND operation as mentioned in SEDSR, so it can expect reduction of execution time overhead. In the case that the signature queue is full, a monitor thread is generated and the “monitor routine” described below is executed on the monitor thread, as depicted in Algorithm 1.

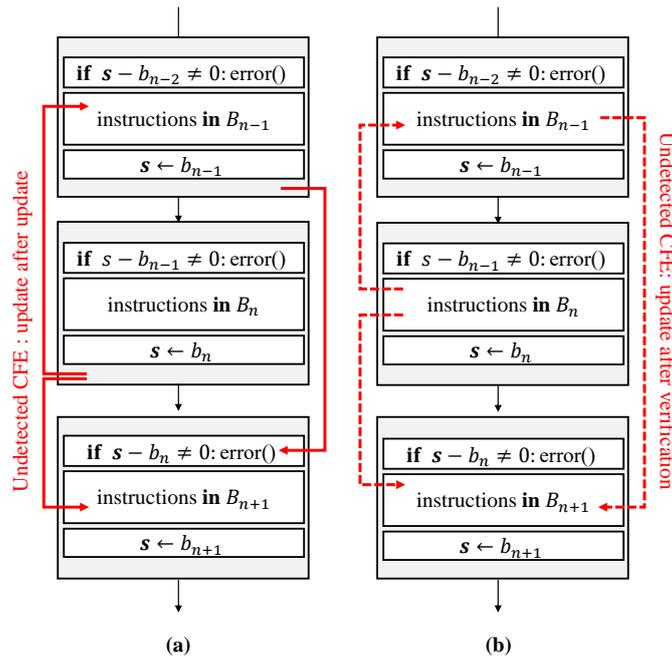


Figure 5. Cases of undetected CFE: (a) update after update; and (b) update after verification.

In Algorithm 1, s is a compile-time signature determined at compile time, which is unique value and represents a basic block. Q_s is a signature queue. \widehat{Q}_s is the current size of signature queue. \widehat{Q}_s^{max} is the maximum size of signature queue. In each basic block, s is enqueued into Q_s . After s is enqueued, \widehat{Q}_s is compared with \widehat{Q}_s^{max} and if \widehat{Q}_s is less than \widehat{Q}_s^{max} , the value of \widehat{Q}_s is incremented by one and signature enqueue routine is terminated. However, if the values of \widehat{Q}_s and \widehat{Q}_s^{max} are equal (signature queue is full), a new thread is created and the monitor routine runs on it. Next, a new signature queue is created and the size of the signature queue \widehat{Q}_s is assigned as zero. Note that the full signature queue is used in the monitor routine.

Algorithm 1: signature_enqueue.

signature_enqueue($s, Q_s, \widehat{Q}_s, \widehat{Q}_s^{max}$)

s : an input signature

Q_s : a signature queue

\widehat{Q}_s : the current size of Q_s

\widehat{Q}_s^{max} : the maximum size of Q_s

generate_thread(function, signature queue) : system function to generate a thread

generate_new_signature_queue(max size) : generate a new signature queue

```

 $Q_s[\widehat{Q}_s] \leftarrow s$ 
if  $\widehat{Q}_s == \widehat{Q}_s^{max}$  then
  generate_new_thread(monitor_routine,  $Q_s$ )
   $Q_s \leftarrow$  generate_new_signature_queue( $\widehat{Q}_s^{max}$ )
   $\widehat{Q}_s \leftarrow 0$ 
end
else
   $\widehat{Q}_s \leftarrow \widehat{Q}_s + 1$ 
end

```

In the non-SSCFM schemes, the run-time signature is shared in a basic block. In other words, the signature update operation is repeated for each basic block in one signature variable. However, in the signature enqueue, a compile-time signature representing each basic block is stored in each place in the signature queue. Therefore, the run-time signature of previous works has a property of being shared in basic block, but run-time signatures in SSCFM are not shared. As indicated, the non-shared characteristics of these run-time signatures can solve the CFE detection rate degradation caused by the run-time signature sharing nature of non-SSCFM schemes

3.4. Monitor Routine and Signature Verification

The “monitor routine” is described in Algorithm 2. The routine runs the signature verification until the signature queue is empty. Algorithm 3 shows a pseudo-code of signature verification performed by comparing the run-time signature in the signature queue and the compile-time signature in the inter-procedural CFG. In the case that the run-time signature in the signature queue does not correspond with the compile-time signature in the inter-procedural CFE, the SSCFM monitor judges that CFE has occurred.

The SSCFM scheme collectively performs signature verification operations in a separate monitor thread. A monitor routine is a collection of signature verification routines that run on separate monitor threads. In Algorithm 2, the signature verification routine is performed until the value of \widehat{Q}_s is zero. There is a signature dequeue routine in the signature verification routine, which dequeues the signatures in the signature queue, decreasing \widehat{Q}_s by one. After all signature verification routines have been completed, the empty signature queue is deleted.

The life cycle of a signature queue is as follows. (1) Generation of signature queue: A signature queue is created in monitor initialization routine at the beginning of the program. The size of the signature queue is determined by the target software application being monitored at compile-stage. (2) Enqueue signature routine: During runtime, the signature enqueue routine is executed in each basic block, and a compile-time signature is accumulated in the signature queue. (3) Full signature queue: If the signature queue is full due to signature enqueue routine, the monitor routine is executed and the monitor routine runs on a separate thread. (4) Delete signature queue: The monitor routine continues until the signature queue is vacated by the signature verification routine, and the empty signature queue is deleted after signature queue is empty. (5) Regeneration of signature queue: Immediately after creating the monitor routine in Step 3, a new signature queue is created. The new life cycle starts from Step 2.

Algorithm 2: monitor_routine.

monitor_routine($Q_s, \widehat{Q}_s, G, v_s$)

Q_s : a signature queue

\widehat{Q}_s : the current size of Q_s

G : inter-procedural CFG

v_s : the node corresponding to signature s in G

v_s : the current node in G

delete_signature_queue(signature queue) : delete the signature queue

while $\widehat{Q}_s \neq 0$ **do**

 | signature_verification($\widehat{Q}_s, Q_s, G, v_s$)

end

delete_signature_queue(Q_s)

In Algorithm 3, *dequeue* routine dequeues the signatures in the signature queue, decreasing \widehat{Q}_s by one. G is the inter-procedural CFG determined at compile-time. G has the connection information of all the basic blocks and function call information. As discussed in more detail in the next subsection, SSCFM has an inter-procedural CFG that represents the connectivity of a basic function block. The nodes of the inter-procedural CFG represent one basic block and each node has a unique compile-time signature. In addition, each node is connected according to the connection of basic block. *get_node_from_signature* routine finds the node corresponding to that signature in the inter-procedural CFG and returns the node to v_s . v_s represents the current node in the inter-procedural CFG, which is the node that updates its state if CFE is not detected in signature verification. *get_next_node_in_graph* routine returns information about the next nodes connected from current node v_s . Next, if v_s is not included in the next nodes, SSCFM judges that CFE has occurred. Otherwise, v_s is updated to v_s .

Algorithm 3: signature_verification.

signature_verification($Q_s, \widehat{Q}_s, G, v_s$)

Q_s : a signature queue

\widehat{Q}_s : the current size of Q_s

G : inter-procedural CFG

v_s : the node corresponding to signature s in G

\hat{v}_s : the current node in G

s : the signature dequeued from Q_s

s_{NEXT} : the set of signatures in the nodes directed by v_s

dequeue(signature queue, the current queue size) : dequeue from Q_s and decrease \widehat{Q}_s by 1

get_node_from_signature(signature) : get v_s from s

get_next_node_in_graph(inter-procedural CFG, the current node)

$s \leftarrow$ dequeue(Q_s, \widehat{Q}_s)

$v_s \leftarrow$ get_node_from_signature(s)

$s_{NEXT} \leftarrow$ get_next_node_in_graph(G, v_s)

if $v_s \notin s_{NEXT}$ **then**

 | error()

end

else

 | $\hat{v}_s \leftarrow v_s$

end

In the non-SSCFM schemes, signature verification is the process of comparing the compile-time signature with the updated run-time signature. Although the SSCFM scheme proposed in this paper has the same semantic relation between the compile-time signature and the run-time signature, in the proposed scheme, the compile-time signature to be compared is in the inter-procedural CFG and run-time signature is in the signature queue. The inter-procedural CFG is generated at the beginning of the program by a monitor initialization routine, which is a routine created through static analysis of the target software application at compile time and is executed only once.

3.5. Inter-Procedural CFG

The SSCFM scheme detects CFE in the coverage of inter-procedural control flow. CFE detection within each function could be one of the factors that degrades the CFE detection performance. Figure 6a illustrates how the SSCFM generates an inter-procedural CFG from each CFG of a function f_1 and a function f_2 .

Each node in inter-procedural CFG represents each basic block in the monitored application, with a compile-time signature that corresponds with the updated run-time signature during runtime. $s_n^{f_m}$ represents the compile-time signature in n_{th} basic block of m_{th} function. The generation of the inter-procedural CFG follows the steps and principles below. (1) The nodes of the control graph in each function represent the basic blocks. (2) Each node is divided into four types: branch, call, entry, and return types. (3) If a node has more than two function calls, the node is divided so that a call-type node has only one function call. (4) The call-type node is connected to the entry type node of the callee function. (5) The return-type node is not connected to any nodes.

In the case of a function call, the connected node is uniquely determined to be one node. However, in the case of a return from a function, it is difficult to uniquely determine the node to be returned: the function can be called by a plurality of functions, and including all the functions in the inter-procedural control graph may cause a drop in CFE detection rates. Therefore, we have introduced a function stack so that the SSCFM scheme uniquely determines the node to be returned in the inter-procedural CFG.

3.6. Function Stack Routine and Function Stack

The “function stack routine” consists of “push” and “pop”, and uses a “function stack” filled with a call-return signature that is shared with a compile-time signature in each call-type and return-type basic block. The function stack is pushed only when the basic block is a call-type. The function stack is popped only when the basic block is a return-type. The function stack routine is included in signature enqueue routine in call-type basic block, not all basic blocks. When the signature queue is full, the function stack is passed to a generated monitor thread. In the monitor thread, the function stack is used when the current node in inter-procedural CFG has to deal with a function call and return, as shown in Figure 6b.

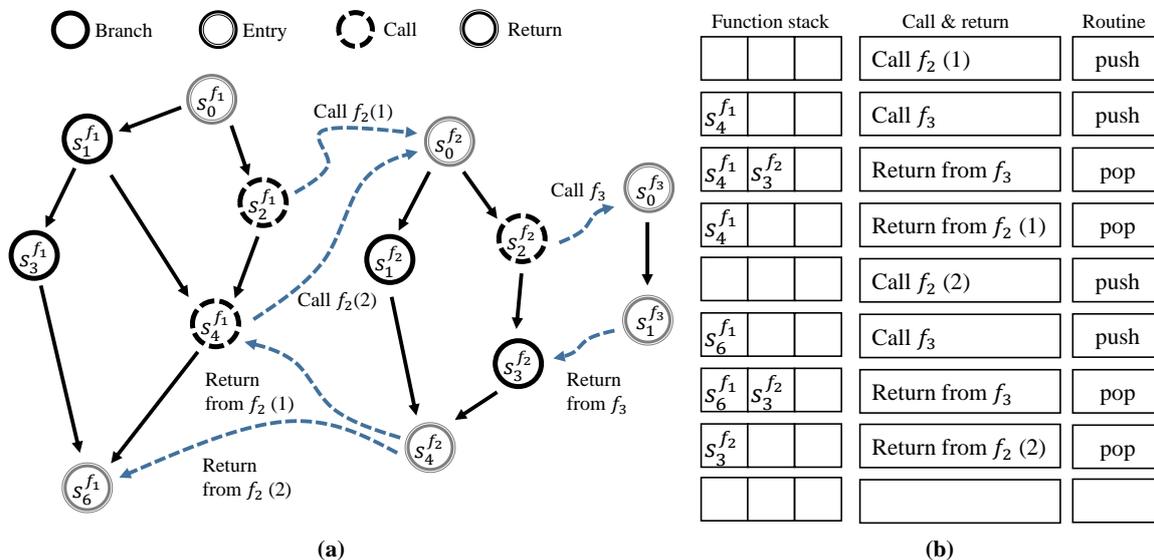


Figure 6. (a) Inter-procedural CFG; and (b) function stack routine.

3.7. Monitor Initialization

The “monitor initialization” routine is inserted at the front of the first basic block of the monitored application. This routine generates the inter-procedural CFG used in the monitor routine and signature verification in the monitor thread, as described above.

3.8. Overall Operation Sequence in SSCFM Scheme

(1) In the compile stage, a static analysis is performed on the target software application for which CFE is to be monitored. At this time, a unique compile-time signature is assigned to each basic block, and an inter-procedural CFG is generated based on the compile-time signature. (2) The signature enqueue routine, the monitor initialization routine, and the monitor routine are also inserted into the target software application at compile time. (3) At the beginning of the program, the monitor initialization routine runs and generates inter-procedural CFG. (4) The program executes a signature enqueue routine for each basic block, and the unique compile-time signatures of each basic block are put into the signature queue. (5) When the signature queue is full, the monitor routine is executed in the new thread and the signature verification routine is executed until the signature queue is empty. Note that signature verification routines exist in the monitor routine. (6) If CFE is not detected until signature queue is empty, the signature queue is removed and the monitor thread is terminated. (7) In Step 5, a new signature queue is created immediately after the monitor routine is created in the new thread. (8) After Step 7, from Step 4 to the end of the program is repeated.

4. Monitoring Scenario with SSCFM Scheme

In this section, we intuitively demonstrate performance improvements with the proposed SSCFM scheme compared with non-SSCFM schemes. The scenario environments are determined by the number of occupable cores (C), the size of the signature queue (Q), and the presence of a blocking I/O of the monitored application (B). Note that, if the non-SSCFM schemes are applied to a system using this multi-threading technique or a multi-core system, execution time overhead will be further increased due to context switching with other threads or other scheduling routines as indicated. This is because the non-SSCFM schemes are based on a single thread.

4.1. Performance Improvements in Multi-Core Environments

The comparison of the non-SSCFM scheme in Figure 4a and the SSCFM scheme in Figure 7a intuitively shows the latter's improved performance compared to the non-SSCFM scheme. There is no signature verification in the application thread, but there is in the monitor thread. Of course, in this case, the number of cores is more than 2 ($C \geq 2$), which presents performance improvements obtained from multi-core environments. However, the non-SSCFM scheme cannot have such performance improvements in multi-core environments.

4.2. Performance Improvements in Multi-Core Environments and Blocking I/O

Figure 7b represents a case of an SSCFM-applied application with blocking I/O in multi-core environments ($C = 1, B = \text{True}$). If the non-SSCFM scheme is applied, the CFE monitoring is blocked along with the application. However, if the SSCFM scheme is applied, the CFE monitoring is able to run in the monitor thread even with the application in a blocked state. Note that the non-SSCFM scheme is based on a single thread. Therefore, if the target application program to be monitored has an I/O operation and is blocked by an I/O operation, the monitor routine will also stop. However, in the SSCFM scheme, a separate thread for CFE monitoring is created and the monitor routine operates on the thread. Therefore, even if the application thread is blocked due to I/O operation, the monitor thread can run in a separate thread.

4.3. Performance Improvements in Multi-Threaded Environments and Blocking I/O

Figure 7c shows a case of an SSCFM-applied application with blocking I/O in an environment in which there is only a single core, or where only one core can be occupied ($C = 1, B = \text{True}$). Even if the application is blocked due to I/O, the monitor thread can be context-switched to keep operating. Of course, there is no guarantee that the monitor thread will always run at that time according to the

scheduling policy. However, since the non-SSCFM scheme is based on a single thread, it is impossible to do so.

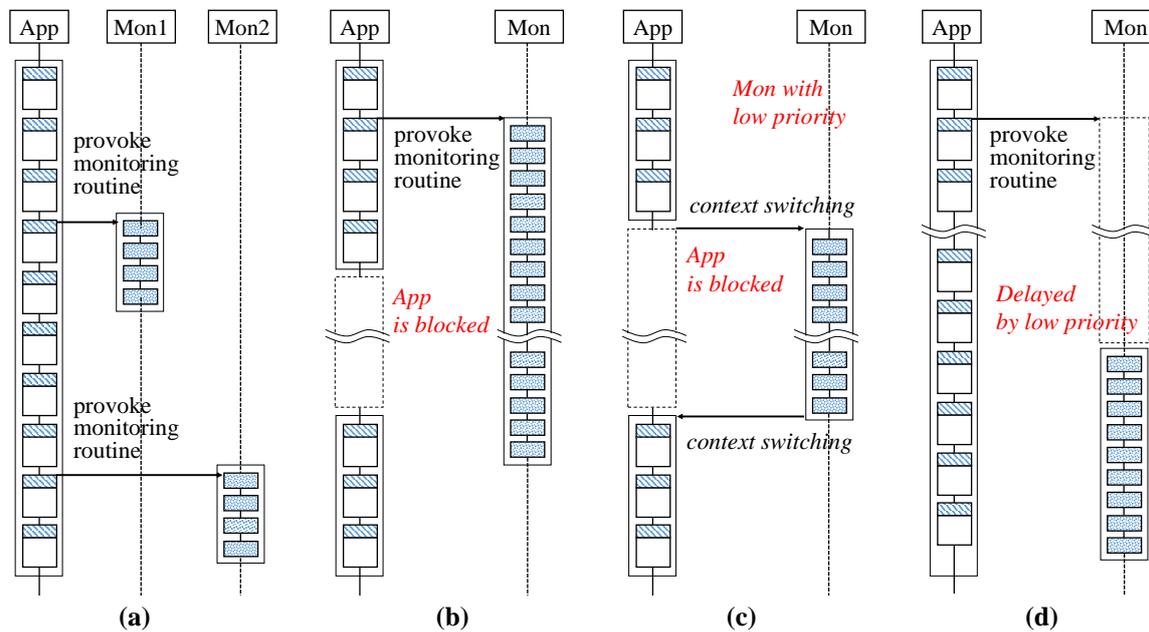


Figure 7. Monitoring scenario with SSCFM: (a) separate signature-based monitoring (SSCFM) scheme ($Q = 4, C \geq 2$); (b) I/O blocking with SSCFM ($Q = 50, C \geq 2, B = \text{True}$); (c) I/O blocking with SSCFM ($Q = 50, C = 1, B = \text{True}$); and (d) delayed CFE monitoring with SSCFM ($Q = 50, C \geq 1$).

4.4. Delayed Monitoring

Figure 7d shows the delayed operation of CFE monitoring in the case that SSCFM scheme is applied. When many applications operate and try to occupy the cores simultaneously, a certain resource may be insufficient. In that case, the SSCFM scheme can lower the priority of the monitor thread so that the monitored application has priority to operate.

5. Code-Generation Framework for SSCFM

This section describes the overall structure of a monitoring code-generation framework for the proposed SSCFM scheme (called an “SSCFM generator”) and shows how each module works. The SSCFM generator depicted in Figure 8 is decomposed into front- and back-end modules. The front-end modules consist of a Clang module, an LLVM IR parser module, and a control flow static analyzer module. The back-end modules consist of a monitoring code generator module, a signature code implanter module, and a target binary code generator module. The front-end modules perform a static analysis of the input application and generate metadata for monitoring code generation, while the back-end modules generate a signature-based monitoring code, insert the monitoring code into the input application, and provide a control flow monitorable application using the SSCFM scheme.

5.1. Clang and LLVM Parser

The SSCFM generator adopts an LLVM compiler infrastructure [28] and LLVM IR in order to analyze a target application. Clang, a well-known module in the LLVM compiler infrastructure, converts the C/C++ code into LLVM IR. The LLVM parser provides a basic block-based data structure to facilitate the basic block-level static analysis.

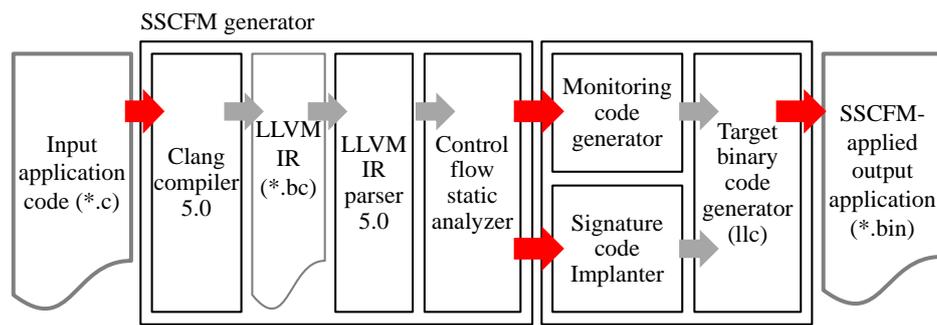


Figure 8. Overall structure of the code-generation framework for the SSCFM scheme.

5.2. Control Flow Static Analyzer

The control flow static analyzer module performs the analysis of the parsed LLVM IR code and generates metadata such as function call graph, the CFG for each function, and compile-time signature in each basic block.

5.3. Monitoring Code Generator

In the SSCFM generator, monitoring code for CFE detection is generated based on metadata from the control flow static analyzer. The monitoring code in the SSCFM scheme consists of monitor initialization code, signature enqueue code, and monitor routine code that includes signature verification. The monitor routine initially generates an inter-procedural CFG and performs signature verification by comparing the inter-procedural CFG with the signature queue in the monitor thread. The code for monitor initialization and monitor routine is generated based on the inter-procedural CFG, and the signature enqueue code is generated based on the compile-time signature in each basic block.

5.4. Signature Code Implanter

The signature code implanter module implants two signature-related codes into a target application; one is code for the monitor initialization routine, and the other is for the signature enqueue routine. The code for the monitor initialization routine is inserted into the first basic block of the target application, and the code for the signature enqueue routine is inserted in every basic block.

5.5. Target Binary Code Generator

The target binary code generator module generates binary code according to the architecture of the system in which the application runs. This module mainly consists of LLVM llc.

6. Performance Evaluation

In this section, we propose the feasibility of performance improvements with the proposed SSCFM scheme compared to one representative non-SSCFM scheme in multi-threaded and multi-core environments.

6.1. Evaluation Environments and Benchmarks

To verify the validity of the proposed SSCFM scheme, its performance evaluation was performed in Ubuntu 16.04.5 Linux, x86-64 architecture with 4 cores and 4 GB RAM memory. Table 1 shows the benchmarks used for performance evaluation and their characteristics. The fast Fourier transform (FFT), Dijkstra (DJ), Patricia (PT), secure hash algorithm (SHA), basic math (BM), stringsearch (SS), and cyclic redundancy check (CRC) provided by Mibench 1.1 [29] were selected as the benchmarks for the performance evaluation, as well as the Whetstone (WS) [30] and Dhrystone (DS) [31].

Table 1. Benchmarks: Whetstone (WS), Dhrystone (DS), fast Fourier transform (FFT), Dijkstra (DJ), Patricia (PT), secure hash algorithm (SHA), basicmath (BM), stringsearch (SS), cyclic redundancy check (CRC).

| No. | Name | Total Number of Functions | Total Number of Basic Block | Number of Instruction Per Basic Block | Total Number of Executed Function Call (k) | Total Number of Executed Basic Block (k) |
|-----|------|---------------------------|-----------------------------|---------------------------------------|--|--|
| 1 | WS | 4 | 77 | 8.08 | 152.9 | 1103 |
| 2 | DS | 12 | 91 | 6.66 | 15.0 | 84 |
| 3 | FFT | 7 | 107 | 6.69 | 32.7 | 4669 |
| 4 | DJ | 6 | 29 | 5.50 | 44.9 | 14,447 |
| 5 | PT | 6 | 212 | 5.40 | 21.7 | 348 |
| 6 | SHA | 8 | 66 | 9.46 | 9.7 | 2558 |
| 7 | BM | 5 | 110 | 5.98 | 121.4 | 11,962 |
| 8 | SS | 10 | 166 | 5.10 | 2.6 | 10,837 |
| 9 | CRC | 4 | 29 | 6.89 | 0.1 | 41,066 |

6.2. Performance Evaluation Criteria and Comparison Target

In the performance evaluation, we tried to compare execution time overhead and CFE detection rate between the proposed SSCFM scheme and the typical non-SSCFM scheme. We selected the SEDSR [14] scheme on behalf of non-SSCFM schemes. The main reasons for selecting the SEDSR scheme as comparison target were as follows. First, we wanted to compare the signature-based CFE monitoring scheme that has the same granularity of CFE detection with our proposed SSCFM scheme. Both our SSCFM scheme and the SEDSR scheme are based on fine-grained CFE detection that performs CFE detection for connectivity of all basic blocks. Second, the SEDSR scheme has the lowest execution time overhead among the fine-grained CFE detection schemes proposed so far. Vankeirsbilck [12] discussed performance comparisons in the same environments for eight representative fine-grained CFE detection schemes, and found the SEDSR scheme had the lowest execution time overhead. Third, the SEDSR scheme is very similar to the proposed SSCFM scheme in terms of operations that deal with signature variables. Most signature-based CFE detection schemes use XOR or AND operations on the signature update routine for signature variables. However, the SEDSR performs the signature update routine by assigning the value of compile-time signatures to run-time signatures. Therefore, it has very low execution time overhead, but low CFE detection rate. The proposed SSCFM in this paper also performs the signature update routine through assignment without any operations such as XOR or AND operations. However, in the SSCFM scheme, the CFE detection rate can be improved by using the signature queue to remove the shared characteristics of the run-time signatures and by fully covering the inter-procedural CFE with the inter-procedural CFG and the function stack. We looked forward to our proposed SSCFM scheme having lower execution time overhead though with quite higher CFE detection rate than SEDSR in multi-threaded and multi-core environments.

6.3. Execution Time Overhead

Figure 9 shows the execution time overhead for both the proposed SSCFM scheme and SEDSR scheme when they were applied to each benchmark. Compared with SEDSR, the execution time overhead with the proposed SSCFM scheme was reduced by 6.79%, 14.71%, 37.67%, 36.88%, 22.01%, 33.34%, 26.00%, 31.06%, and 31.55% of performance improvements for WS, DS, FFT, DJ, PT, SHA, BM, SS, and CRC, respectively. The average execution time overhead was approximately 26.67%. This performance improvement in execution time overhead reduction came from both multi-core and multi-threading techniques.

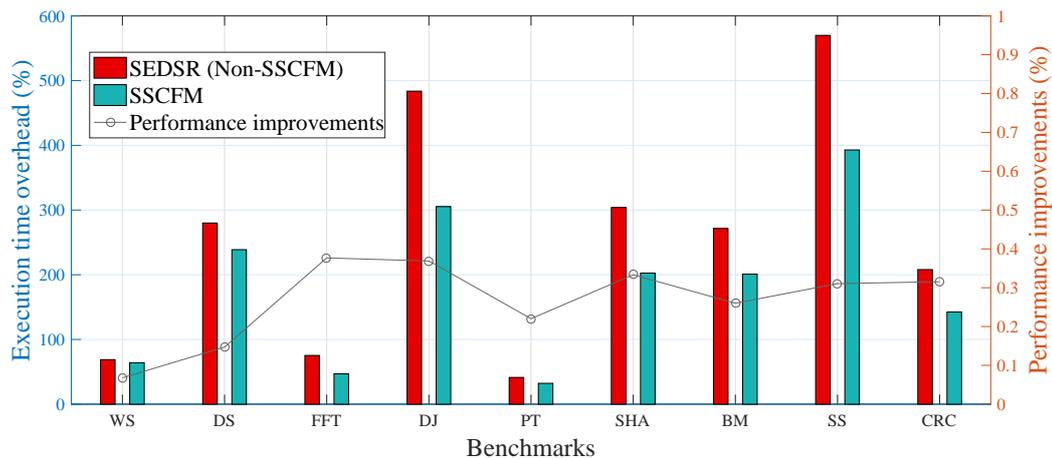


Figure 9. Execution time overhead.

6.4. Execution Time Overhead Analysis

We analyzed how the execution time overhead was reduced compared to the non-SSCFM schemes. Figure 10 describes the percentage of execution time overhead for four monitoring routines: signature enqueue, function stack routine, monitor initialization, and thread generator. The average executive time overheads were 91.126%, 7.932%, 0.577%, and 0.368% for signature enqueue, function stack routine, monitor initialization, and thread generator, respectively. In the SSCFM scheme, signature update is replaced by signature enqueue, and there are additional routines, such as function stack routine, monitor initialization, and thread generator. However, with the exception of signature enqueue, the total execution time overhead only took 8.87% on average. Given that no execution time overhead is required for signature verification in the SSCFM scheme, this naturally led to an overall reduction in execution time overhead therein.

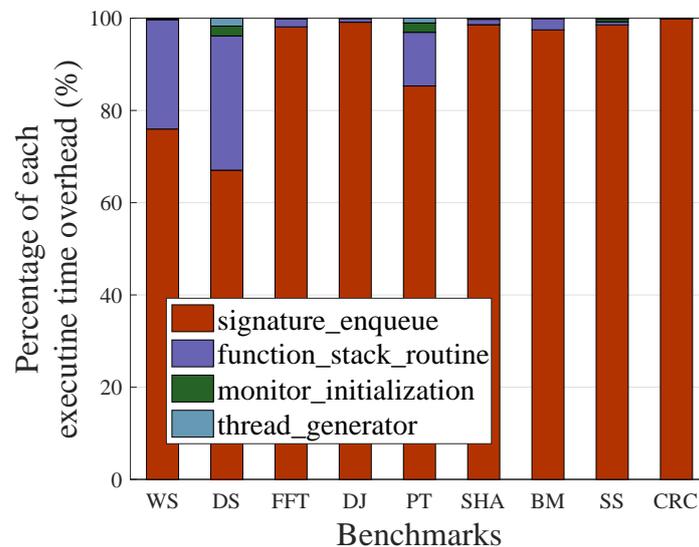


Figure 10. Percentage of each execution time overhead.

In Figure 9, the benchmarks are divided into two groups. The first group is comprised of FFT, DJ, SHA, BM, SS, and CRC, with performance improvements in execution time overhead of 37.67%, 36.88%, 33.34%, 26.00%, 31.06%, and 31.55%, respectively. The second group is WS, DS, and PT, showing performance improvements in execution time overhead of 6.79%, 14.71%, and 2.01%, respectively.

The major reason behind the performance differences can be found in Figure 10. In the first group, the percentage of execution time overhead for the function stack routine is as high as 21.48% on average. However, in the second group, the percentage of execution time overhead for the function stack routine is very low, only 1.15% on average. The execution time overhead for the function stack routine increased with the percentage of the total number of executed function calls in the total number of executed basic blocks. These data can be also found in Table 1.

6.5. CFE Detection Rate

Figure 11 shows the CFE detection rate with the proposed SSCFM scheme and SEDSR scheme applied to each benchmark. The average CFE detection rate of SEDSR was approximately 49.97%. The undetected case of CFEs in SEDSR scheme was caused by the shared characteristics of run-time signature, as noted in Section 3.2. The multi-staged signature update or signature verification proposed in the non-SSCFM schemes could solve this problem, but this approach increased the execution time overhead. The SSCFM scheme solved the problem with the signature queue, improving the CFE detection rate to 91.18%, 89.67%, 96.45%, 90.96%, 89.44%, 87.60%, 97.52%, 97.63%, and 96.77% for WS, DS, FFT, DJ, PT, SHA, BM, SS, and CRC, respectively, without increasing execution time overhead. The average CFE detection rate of SSCFM was approximately 93.69%.

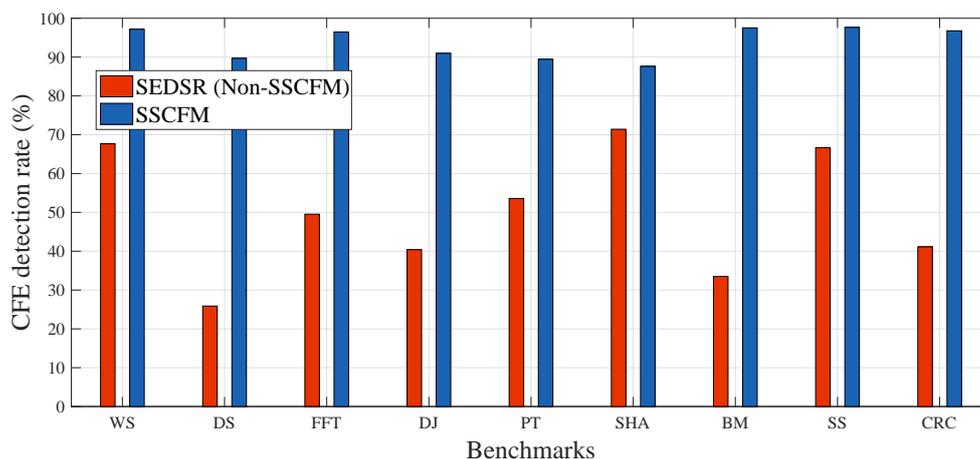


Figure 11. CFE detection rate.

7. Conclusions

Multi-threading techniques and multi-core environments have made overall improvements to computing systems and are now widely used in almost all computer systems. However, considering previously proposed several schemes, there is still a part that cannot improve performance in multi-threaded and multi-core environments. Rather, performance is degraded in multi-threaded and multi-core environments in some case. In this paper, we introduce the SSCFM scheme that enables performance enhancement in multi-threaded and multi-core environments by modifying the previously proposed signature-based CFE monitoring schemes. By separating signature update routines and signature verification routines into thread levels in the signature-based CFE monitoring scheme, we look forward to reducing the execution time overhead of the SSCFM scheme. Compared with SEDSR, a very similar type of scheme that operates on a single thread, the execution time overhead with the proposed SSCFM scheme is reduced to approximately 26.67% on average of that of SEDSR. Additionally, by using a signature queue that removes the shared characteristics of the run-time signature and by fully covering the inter-procedural CFE with the inter-procedural CFG and the function stack, the average CFE detection rate of the SSCFM scheme can be increased to 93.69%. In addition, this paper introduces SSCFM generator, which can automatically apply SSCFM scheme

to compiler base. Due to the nature of software-based CFE monitoring, the monitoring code must be inserted into each basic block. Even if it has a good performance scheme, it would be difficult to apply it if it is applied by hand. In future work, we will try to optimize the proposed SSCFM scheme to have lower execution time overhead.

Author Contributions: K.C. designed the entire core architecture and performed the hardware/software implementation and experiments; J.C. proposed key concept and algorithm of the proposed architecture as the first corresponding author; and D.P. has responsibility as the second corresponding author.

Funding: This study was supported by the BK21 Plus project funded by the Ministry of Education, Korea (21A20131600011). This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2014R1A6A3A04059410 and 2016R1D1A1B03934343).

Conflicts of Interest: The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---------|--|
| CFE | Control Flow Error |
| CFG | Control Flow Graph |
| SSCFM | Separate Signature-based Control Flow Error Monitoring |
| LLVM | Low Level Virtual Machine |
| LLVM IR | LLVM Intermediate Representation |
| FFT | Fast Fourier Transform |
| DJ | Dijkstra |
| PT | Patricia |
| SHA | Secure Hash Algorithm |
| BM | Basic Math |
| SS | String Search |
| CRC | Cyclic Redundancy Check |
| WS | Whetstone |
| DS | Dhrystone |

References

- Baumann, R. Soft errors in advanced computer systems. *IEEE Des. Test Comput.* **2005**, *22*, 258–266. [[CrossRef](#)]
- Baffreau, S.; Bendhia, S.; Ramdani, M.; Sicard, E. Characterisation of microcontroller susceptibility to radio frequency interference. In Proceedings of the Fourth IEEE International Caracas Conference on Devices, Circuits and Systems (Cat. No.02TH8611), Aruba, Dutch Caribbean, 17–19 April 2002; p. 1031. [[CrossRef](#)]
- Hu, T.; Guo, Z.; Yi, P.; Baker, T.; Lan, J. Multi-controller Based Software-Defined Networking: A Survey. *IEEE Access* **2018**, *6*, 15980–15996. [[CrossRef](#)]
- Feng, C.; Lu, Z.; Jantsch, A.; Zhang, M.; Xing, Z. Addressing Transient and Permanent Faults in NoC with Efficient Fault-Tolerant Deflection Router. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2013**, *21*, 1053–1066. [[CrossRef](#)]
- Raha, P.; Vinodhini, M.; Murty, N.S. Horizontal-vertical parity and diagonal hamming based soft error detection and correction for memories. In Proceedings of the 2017 International Conference on Computer Communication and Informatics (ICCCI), Nicosia, Cyprus, 27–29 September 2017; pp. 1–5. [[CrossRef](#)]
- Liu, S.; Reviriego, P.; Xiao, L. Evaluating Direct Compare for Double Error-Correction Codes. *IEEE Trans. Device Mater. Reliab.* **2017**, *17*, 802–804. [[CrossRef](#)]
- Li, T.; Ambrose, J.A.; Ragel, R.; Parameswaran, S. Processor Design for Soft Errors: Challenges and State of the Art. *ACM Comput. Surv.* **2016**, *49*. [[CrossRef](#)]
- Didehban, M.; Shrivastava, A.; Lokam, S.R.D. NEMESIS: A software approach for computing in presence of soft errors. In Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 13–16 November 2017; pp. 297–304. [[CrossRef](#)]

9. Mitropoulou, K.; Porpodas, V.; Jones, T.M. COMET: Communication-optimised Multi-threaded Error-detection Technique. In Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, Pittsburgh, PA, USA, 1–7 October 2016. [\[CrossRef\]](#)
10. Chen, Z.; Nicolau, A.; Veidenbaum, A.V. SIMD-based Soft Error Detection. In Proceedings of the ACM International Conference on Computing Frontiers, Sardinia, Italy, 30 April–2 May 2016; pp. 45–54. [\[CrossRef\]](#)
11. Nicolescu, B.; Velazco, R. Detecting soft errors by a purely software approach: Method, tools and experimental results. In Proceedings of the 2003 Design, Automation and Test in Europe Conference and Exhibition, Munich, Germany, 3–7 March 2003; pp. 57–62. [\[CrossRef\]](#)
12. Vankeirsbilck, J.; Penneman, N.; Hallez, H.; Boydens, J. Random Additive Signature Monitoring for Control Flow Error Detection. *IEEE Trans. Reliab.* **2017**, *66*, 1178–1192. [\[CrossRef\]](#)
13. Asghari, S.A.; Taheri, H.; Pedram, H.; Kaynak, O. Software-Based Control Flow Checking Against Transient Faults in Industrial Environments. *IEEE Trans. Ind. Inform.* **2014**, *10*, 481–490. [\[CrossRef\]](#)
14. Asghari, S.A.; Abdi, H.T.A.; Pedram, H.; Pourmozaffari, S. SEDSR: Soft Error Detection Using Software Redundancy. *Sci. Res.* **2012**, *5*, 664–670. [\[CrossRef\]](#)
15. Li, A.; Hong, B. Software implemented transient fault detection in space computer. *Aerosp. Sci. Technol.* **2007**, *11*, 245–252. [\[CrossRef\]](#)
16. Oh, N.; Shirvani, P.P.; McCluskey, E.J. Control-flow checking by software signatures. *IEEE Trans. Reliab.* **2002**, *51*, 111–122. [\[CrossRef\]](#)
17. Goloubeva, O.; Rebaudengo, M.; Reorda, M.S.; Violante, M. Soft-error detection using control flow assertions. In Proceedings of the 18th IEEE Symposium on Defect and Fault Tolerance in VLSI Systems, Boston, MA, USA, 3–5 November 2003; pp. 581–588. [\[CrossRef\]](#)
18. Alkhalifa, Z.; Nair, V.S.S.; Krishnamurthy, N.; Abraham, J.A. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. Parallel Distrib. Syst.* **1999**, *10*, 627–641. [\[CrossRef\]](#)
19. Bondy, J.; Murty, U. *Graph Theory*, 1st ed.; Springer Publishing Company: New York, NY, USA, 2008.
20. Vemu, R.; Gurusurthy, S.; Abraham, J.A. ACCE: Automatic correction of control-flow errors. In Proceedings of the 2007 IEEE International Test Conference, Santa Clara, CA, USA, 21–26 October 2007; pp. 1–10. [\[CrossRef\]](#)
21. Khudia, D.S.; Mahlke, S. Low Cost Control Flow Protection Using Abstract Control Signatures. *SIGPLAN Not.* **2013**, *48*, 3–12. [\[CrossRef\]](#)
22. Dietrich, C.; Hoffmann, M.; Lohmann, D. Global Optimization of Fixed-Priority Real-Time Systems by RTOS-Aware Control-Flow Analysis. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*. [\[CrossRef\]](#)
23. Zhang, M.; Gu, Z.; Li, H.; Zheng, N. WCET-Aware Control Flow Checking with Super-Nodes for Resource-Constrained Embedded Systems. *IEEE Access* **2018**, *6*, 42394–42406. [\[CrossRef\]](#)
24. Tullsen, D.M.; Eggers, S.J.; Levy, H.M. Simultaneous multithreading: Maximizing on-chip parallelism. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, Ligure, Italy, 22–24 June 1995; pp. 392–403.
25. Blake, G.; Dreslinski, R.G.; Mudge, T. A survey of multicore processors. *IEEE Signal Process. Mag.* **2009**, *26*, 26–37. [\[CrossRef\]](#)
26. Khoshavi, N.; Zarandi, H.R.; Maghsoudloo, M. Two control-flow error recovery methods for multithreaded programs running on multi-core processors. In Proceedings of the 2012 28th International Conference on Microelectronics Proceedings, Nis, Serbia, 13–16 May 2012; pp. 371–374. [\[CrossRef\]](#)
27. Zhu, Z.; Callenes-Sloan, J. Towards low overhead control flow checking using regular structured control. In Proceedings of the 2016 Design, Automation Test in Europe Conference Exhibition (DATE), Dresden, Germany, 14–18 March 2016; pp. 826–829.
28. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. In Proceedings of the International Symposium on Code Generation and Optimization, CGO 2004, San Jose, CA, USA, 20–24 March 2004; pp. 75–86. [\[CrossRef\]](#)
29. Guthaus, M.R.; Ringenberg, J.S.; Ernst, D.; Austin, T.M.; Mudge, T.; Brown, R.B. MiBench: A free, commercially representative embedded benchmark suite. In Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization, Austin, TX, USA, 2 December 2001; pp. 3–14. [\[CrossRef\]](#)

30. Curnow, H.J.; Wichmann, B.A. A synthetic benchmark. *Comput. J.* **1976**, *19*, 43–49. [[CrossRef](#)]
31. Weicker, R.P. Dhrystone: A synthetic systems programming benchmark. *Commun. ACM* **1984**, *27*, 1013–1030. [[CrossRef](#)]



© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).